Working Paper Series
ISSN 1170-487X

Constructing Integrated Software Development Environments with Dependency Graphs

by John C. Grundy and John G. Hosking

Working Paper 94/4
March, 1994

Department of Computer Science
The University of Waikato
Private Bag 3105
Hamilton, New Zealand

Constructing Integrated Software Development Environments with Dependency Graphs

John C. Grundy and John G. Hosking*

Abstract

Integrated software development environments need to support multiple textual and graphical views of software products under development. MViews, a new model for constructing such environments, provides abstractions for representing the abstract syntax of a program as graphs and for viewing and manipulating these graphs in concrete textual and graphical forms. Graphs are used to represent software system structures using components (graph nodes) and relationships (graph edges). These graph components are modified by graph operations to construct and modify a program. Views of these graph components are rendered and manipulated in either graphical or textual forms.

Consistency management between updated graph components is supported by a novel update record mechanism. MViews graphs are dependency graphs: updates to graph components are broadcast to related components as update records. These dependent components, or the inter-connecting relationships, respond to these update records and update the dependent components appropriately. This mechanism is used to support inter-component consistency management, and in particular provides a novel way keeping textual and graphical views of software development consistent. This mechanism can also be used as the basis of a wide range of other software development environment facilities. MViews has been reused to implement a variety of integrated environments and examples of these environments are discussed.

1. Introduction

Programming environments assist programmers to implement and debug programs by providing tools which make the task of program construction easier [11]. Integrated software development environments (ISDEs) subsume programming environments and provide tools for various software management tasks such as analysis, design, implementation, debugging, maintenance and version control [26, 11]. ISDEs usually require both graphical and textual representations of *parts* of a software system, and each such partial representation is called a *view*. Support for many views and view types is usually needed in an ISDE together with some form of consistency management between the different views and software representations that share information. A mechanism that permits new or existing tools to be integrated into the environment is also essential [26]. In this paper we describe MViews, a new model and framework for constructing ISDEs which provides such support.

MViews represents the structure of software systems and their views as graphs, which can be extended with static language semantic values using further graph components and relationships. The graphs are organised in three layers. A *base layer* provides a representation of shared information. *View layers* provide view specific representations and behaviour, while *display layers* render, and permit manipulation of, view layer information either graphically or textually. Consistency between graph components is supported by a novel update record propagation mechanism. This uses the arcs in the graph to propagate a record of the exact change to a component to other components dependent on its state. Base and view components are kept consistent via this mechanism and this allows graphical and textual view representations to be kept fully consistent, no matter which kind of view is manipulated. External tools can also be interfaced to MViews environments via views.

We begin by surveying current approaches to the development of ISDEs. MViews is then described and illustrated with examples from systems developed using the model. Development of environments using MViews is discussed and experience of building several systems using this model is presented. Current and future research on MViews and its derivatives is summarised.

---

* Author's addresses: J. Grundy, Department of Computer Science, University of Waikato, Private Bag 3105, Hamilton, New Zealand; J. Hosking, Department of Computer Science, University of Auckland, Private Bag, Auckland, New Zealand.

2. Related Research

Traditional programming environments are based on an edit-compile-run sequence of program development [11]. Such environments usually support text-based program development with limited graphical user interfaces and with tools loosely integrated using a file system [26]. Program data, views and tools do not usually have a consistent, shared representation mechanism. Thus consistency management between graphical tools and interfaces and textual program representations either does not exist or is unidirectional from text to graphical forms [32]. Similarly, with purpose-built environments, such as THINK C on the Macintosh [35], and language-based environments, such as Smalltalk [15], information in graphical and textual views is either non-overlapping or has limited consistency management. Graphical views are usually regenerated as required (for example, class browsers and dialogs are simply re-rendered). These approaches mean that very limited amounts of over-lapping information can be supported before view inconsistencies arise. Abstract models and reusable frameworks for constructing such multi-view editing environments are often simplistic and over-general, such as the MVC model for Smalltalk [15] (see later discussion).

CASE tools, such as Software thru Pictures [37] and the OOATool [8], provide graphical editors supporting the construction of analysis and design diagrams. These usually provide consistency management between different graphical views. Most CASE tools do not support program implementation although some generate program fragments from a design and allow programmers to incorporate these into their own programs. A major drawback of this approach is a lack of consistency management and traceability between design and implementation, leading to problems if one or both are modified independently. Reverse-engineering of design diagrams from program code partially solves this problem by allowing design information to be regenerated (or possibly modified) to reflect program change. This does not solve the problem of design change impacting on programs, however, nor does it allow "fuzzy" changes to be propagated (i.e. changes that can not be directly translated from design to implementation representations).

Visual programming environments, such as Fabrik [20] and Prograph [9], must keep different graphical views of programs consistent. This is straightforward when views contain information that is directly updateable by changes applied to other views. An environment supporting abstract design and visual programming views, however, must provide a mechanism for translating changes in one type of view to appropriate changes in other types of view. These changes may not necessarily be directly applicable to other kinds of views, for example an entity-relationship connection in a design view modelled as a dataflow in a visual program view.

Many researchers have attempted to support declarative specification and generation of languages and their environments. These have usually been based on the abstract syntax of a language together with automatic generation of structure-oriented editors. Examples include the Cornell Program Synthesizer [34] and Mjølner environments [25]. These systems allow language structure and static semantics to be specified abstractly and in a declarative manner. Environments produced in this way are generally not very extensible, however, and have a restrictive editing style [38]. As they are text-based, they require additional tools for software analysis and design which usually introduces problems with data and user interface consistency and integration.

Dora [29] and PECAN [30] use structure-oriented editing techniques on a shared program representation to support view consistency by propagating editing changes between views. A weakness is that changes not directly able to be applied to another view can not be indicated to programmers. FormsVBT [4] supports multi-view editing with consistency via token substitution in textual views and incremental redisplay of graphical views. This approach means graphical view updates must be locked out when a textual view is edited, however, and only a simple S-expression language can be supported for the textual program. Systems such as Zeus [6], Dora [29], and GLIDE [22] provide some abstractions for building multi-view editing environments. In general, however, the support for ISDE view consistency management they provide is too inflexible. Semantic Program Graphs (SPGs) [27] use a hypergraph-based notation to represent both the executable and unexecutable aspects of software systems for multi-view development environments. While a flexible approach to base software system specification, SPGs do not directly describe how multiple textual and graphical views of a software system are defined and manipulated, nor how these views are kept consistent under change.

Multiple textual and graphical views of information are common to most ISDEs and development of these systems is a large programming effort. An appropriate set of abstractions for constructing such environments would thus be very useful. In the above work, different view and tool integration

mechanisms, including file system integration, database and database view integration, and canonical form representation, have been attempted with varying degrees of success [26]. In the development of MViews we have aimed for a homogeneous solution which provides a useful set of ISDE abstractions based on a uniform conceptual model of dependency graphs. This model can be used as a set of reusable building-blocks, or *framework,* for more easily constructing new environments and tools.

3. SPE

To focus the following detailed description of MViews, we first introduce SPE (the Snart Programming Environment) an ISDE for Snart, an object-oriented extension to Prolog. SPE is implemented using the MViews framework, and will serve as an example environment throughout our discussion of MViews. A more detailed description of SPE may be found in [18]. Other MViews-based environments will be described in later sections.

Fig. 1. shows a screen dump from SPE during the development of a drawing editor program. Several windows are shown, each corresponding to different views of the program. One graphical view shows an analysis-level diagram representing important generalisation and aggregation structures. The second shows a design-level diagram describing method calling protocols when rendering figures in a drawing window. One textual view shows a detailed class interface for the `drawing_window` class, another shows a method implementation, and the third shows detailed documentation about the window class.

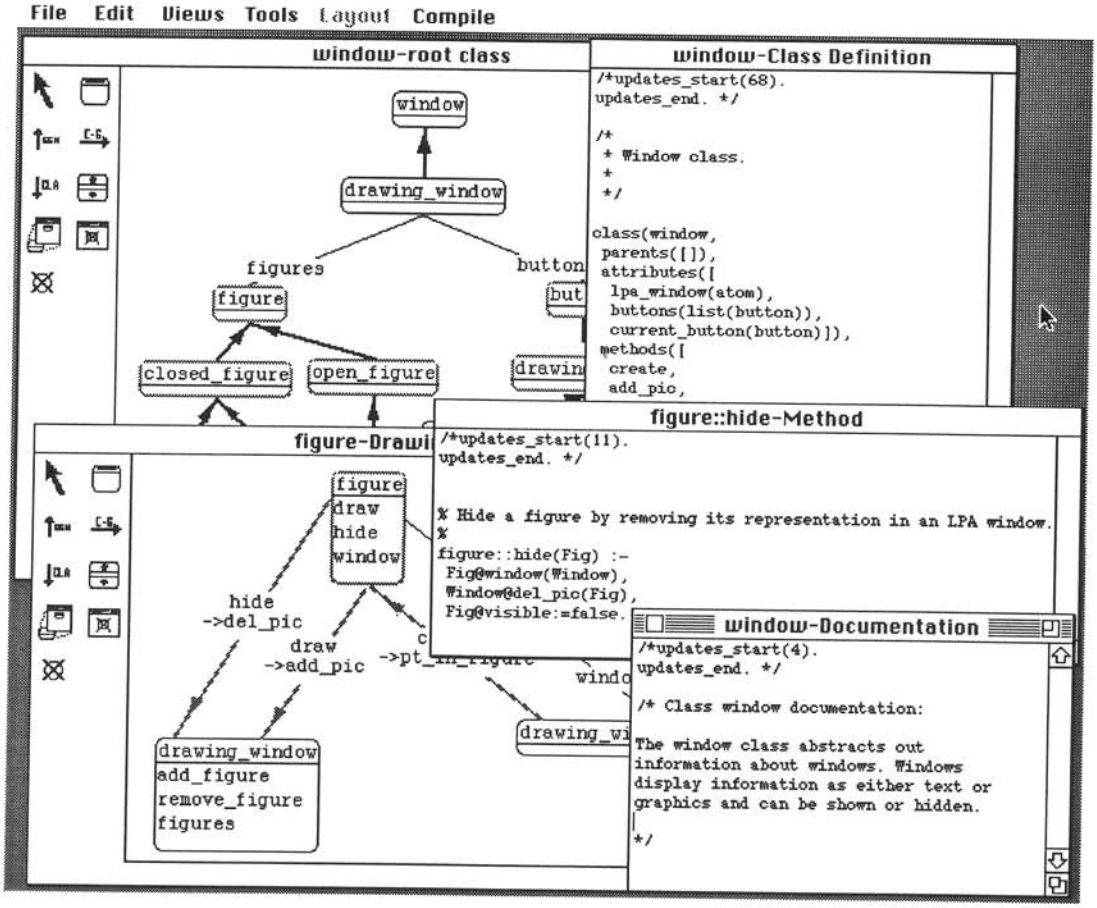


Fig. 1. An example of SPE views during the development of a simple drawing program.

There is no restriction on the number and types of views able to be constructed and displayed, and the contents and layout of each view are under user control. Graphical views include a palette, used for selecting tools for interactive manipulation of the view contents (which use an icon and connector representation). Textual views are free-edited. Inter-view navigation is either via menus or via an automatically constructed point and click type hyper-link system, which allows access from any view component to any other view also containing that component. As there is over-lapping information in each kind of view, SPE must keep these different views consistent under change. To achieve this it ensures:

- changes are propagated to all affected views (so no inconsistent information is manipulated)

- where appropriate, views are updated automatically (so programmers need not be concerned with doing this themselves)
- changes made to one view that can't be automatically applied to other affected views (eg propagation of an analysis view change to a code view) are indicated in some way (so programmers are aware of inconsistencies and know to update a view themselves)

4. The MViews Model for ISDEs

The MViews framework provides support for the following abstractions: flexible representation of software structure and static language semantics; definition of different views and view representations, both textual and graphical; view and component consistency management; simple specification of editor functionality and user interfaces; persistency; multi-user development; and tool integration mechanisms. This is by no means an exhaustive collection of abstractions useful for ISDE construction; built-in support for version control would be useful, for example. However, it provides most of the necessary elements in a highly reusable framework.

4.1. Software System Representation

An environment framework must be capable of constructing environments for a diverse range of languages and system representations. Thus software system structure needs to be represented in a manner which is flexible. The representation also needs to be close to an environment's needs (i.e. a "natural" representation for the application domain) [26, 3]. This latter requirement may, of course, conflict with the former. A complementary mechanism for describing language-specific semantics is also required [34, 5]. MViews adopts a graph-based approach for representing both the abstract syntax of software structures and static semantic values associated with these structures. This is appropriate for most of the program structures an ISDE should model, including graph-based visual languages which plain abstract syntax trees cannot model [5]. Another advantage over ordinary abstract syntax structures is flexibility of construction: graph components may be built up independently and then combined via appropriate links.

MViews represents environment data as a collection of (possibly disjoint) directed graphs. Software components are represented as *components* (nodes) and are connected by *relationships* (labelled edges). Each component has *attributes* (name/value pairs) associated with it (symbolically represented by the quoted text items in Fig. 2.). Some relationships are simple in that they just link related components while others contain information about the relationship as attributes. For example, the *type-of* relationship in Fig. 2 has an associated attribute specifying the feature *figures* is of type *list figure* rather than just *figure*. Relationships also behave as components i.e. they can be connected by other relationships, and thus we collectively refer to all graph components (nodes and edges) as components. Fig. 2. shows an example of an MViews graph for part of a drawing editor program modelled in SPE.

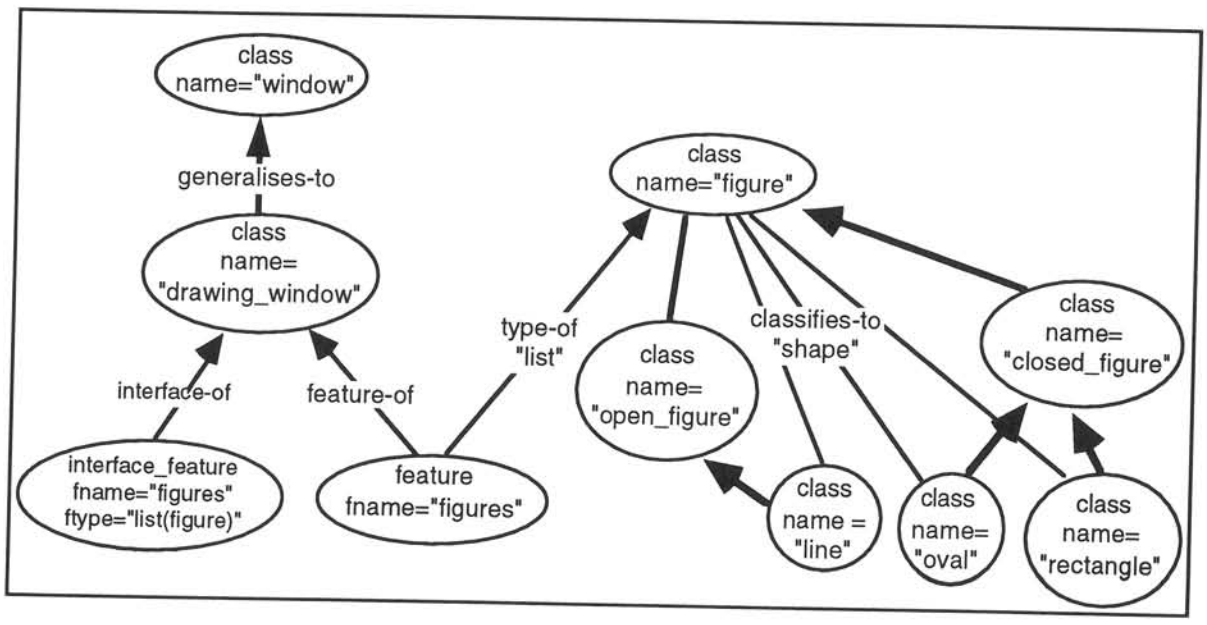


Fig. 2. Part of the program graph for an object-oriented design of a drawing program.

In addition to the structure of a software system, values representing some of the computed semantic meanings of these structures can be associated with components. These values are represented using the same graph-based form (for example, the computed interface of a class in fig. 2), and equate to the attributes of decorated abstract syntax trees used in attribute grammar systems [19, 34].

4.2. Views and View Representation

Views model a subset of the total software system state and provide a textual or graphical rendering of this "partial system". Views can also be thought of as editing interfaces to the underlying software system. Structures need to be modified to develop a software system and in the programmer's mind, modification of a view is equivalent to changing the part of a system displayed in the view [26, 36]. View editing operations are thus translated into appropriate software system modifications. Vlissides [90] argues that the structure of views should be similar to the structure of base software data, to allow view data to be manipulated in a similar manner to the base data it mirrors. There may, however, be some scope for structuring views differently for efficiency or because a different structure is a more appropriate model for the view [10].

MViews uses a three-layer architecture to achieve these goals. Fig. 3 shows an example of this three layer view architecture as used in the implementation of SPE. A single *base layer* provides a shared representation of the software system as a graph. *View layers* are graphs representing the information needed for each view, i.e. base layer elements and relationships have corresponding view layer elements and relationships. In Fig. 3, three such view layers are shown. There, base layer "class" components are represented by view layer "class icon" components. *Display layers* act as both renderers and interactive editors of the view layer components with which they are associated.

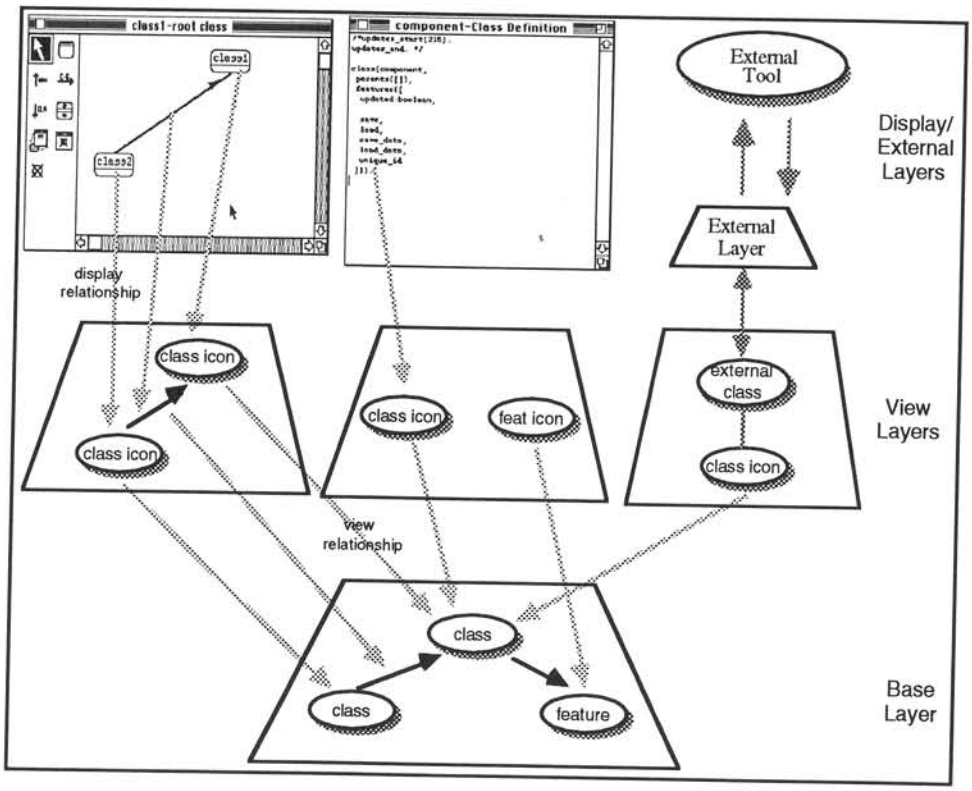


Fig. 3. Typical program and view relationships in the MViews implementation of SPE.

View layer components (view components) are usually connected to base layer components (base components) via inter-view relationships (view relationships in Fig. 3). These allow view components to access base component data, and permit changes to be propagated bi-directionally between a base component and its view components. View components, however, need not always be connected (mapped) to a base component. This allows partial, but controlled, inconsistency at the view level. It also provides a mechanism for temporarily retaining view components when their base components have been deleted. This mechanism could be used, for example, to remap a disconnected view component to another base component.

Base components typically have more than one view component in one or more views. Each view component represents a subset of its corresponding base view component's attributes and relationships. View components typically model one base component although they may be mapped to, and hence model, more than one base component. This allows composite components to be represented in views. For example a "feature icon" for SPE might have a class and feature name and be mapped to the base class and base feature at the same time (so it can respond to changes in both base components). View components may also be chained together to abstract away from specific kinds of base components, forming a flexible view component abstraction mechanism.

Each view layer can be rendered (displayed) either graphically or textually using an appropriate *display layer*. A display layer renders a view component in a textual or graphical form, and connects this rendering to the view component via a *display* relationship. The display layer supplies both a rendering and an editing mechanism so view components can be both visualised and modified by programmers. MViews supports a variety of view editing approaches including free form editing of textual views and direct manipulation for graphical views. Programmers generally find these approaches to be more natural to use than comparable structure-editing approaches [3, 38, 39], as used by most other integrated environments [25, 29].

Editing changes are applied to view components, and are thence propagated to base components. Display layer renderings are regenerated (possibly incrementally) when their view component changes. Two display layers are shown in Fig. 3, one textual and one graphical. In the graphical view, the view components are rendered as class icons and generalisation and client-supplier glue (icon connectors).

4.3. Tool Integration and Extensibility

Environments are typically made up of several tools used for different purposes, for example editing, compiling, debugging and version control. Environment integration should be at both the user interface level (providing a consistent user interface across all tools) and the tool data level (providing uniform data storage or translation mechanisms) [26, 32]. This is really an extension of the multiple view support requirements described earlier, but where "tool views" need to be supported. An environment should also be extensible, allowing new tools to be developed or existing tools from other systems to be integrated in a consistent manner [26, 32].

Views can also be used for tool extensibility and integration. The view layer provides a tool-specific interface to the canonical program structure stored in the base view. Display layers and dialogues provide a data mapping facility for exporting and importing external tool data, using parsing and unparsing, and tool user interface integration, as shown by the "external view" of Fig. 3., in a similar manner to FIELD environments [33].

4.4. Program and View Modification

A more formal description of the state of an MViews environment, i.e. its graph, G, is the 4-tuple:
$G \stackrel{def}{=} \langle C,R,A,V \rangle$

where:

- C is the set of components that comprise the graph, $C \stackrel{def}{=} Component \rightarrow ComponentKind$, where *Component* is unique for G and *ComponentKind* denotes the kind of program component (which also gives component-specific macro operations, i.e. $ComponentKind \rightarrow Op \rightarrow Args \rightarrow G \rightarrow G$, where $Op \stackrel{def}{=} String$, $Args \stackrel{def}{=} \textbf{seq}\ Value$);
- R is a connection relation over the components of the graph, $R \stackrel{def}{=} RelComp \rightarrow Parent \rightarrow Child$, where $RelComp \stackrel{def}{=} C$, $Parent \stackrel{def}{=} C$, $Child \stackrel{def}{=} C$;
- A gives the attributes of each component, $A \stackrel{def}{=} Component \rightarrow AttributeName \rightarrow Value$, where $AttributeName \stackrel{def}{=} String$ and $Value \stackrel{def}{=} Boolean|Integer|String|List(Value)$;
- V is the set of views, $V \stackrel{def}{=} C \rightarrow Elements$, where $Elements \stackrel{def}{=} \{C\}$.

This graph defines the state of an MViews environment. Manipulation and modification of this state is via graph *operations*. MViews components support a number of fundamental graph operations, shown in Table 1. These fundamental operations (and their meanings) are used in the following section on consistency management in MViews environments. *Macro operations*, built up from a sequence

fundamental (and other macro) operations, can also be defined for, and associated with, each kind of component.

Operation	New Graph	Description
AddComp(Kind,Comp)	C' ← C ∪ {Comp→Kind}	Add a new component to the graph
EstRel(Kind,Parent,Child,Rel)	R' ← R ∪ {Rel→Parent→Child} ; AddComp(Kind,Rel)	Establish a relationship between Parent and Child components
CreateView(Kind,View)	V' ← V ∪ {View→∅} ; AddComp(Kind,View)	Create a new view
DeleteComp(Comp)	C' ← C ψ[1] Comp ; A' ← A ψ Comp ; ∀((Rel→Comp→Child ∈ R ∨ Rel→Child→Comp ∈ R)• DissolveRel(Rel)) ; ∀View∈ V ∧ Comp∈ V(View)• RemoveFromView(View,Comp)	Delete a component from the graph. Must remove component kind, attributes, dissolve relationships to other components, and remove from view.
DissolveRel(Rel)	R' ← R ψ Rel ; DeleteComp(Rel)	Dissolve a relationship
DeleteView(View)	∀Comp∈ V(View)• DeleteComp(P,Comp)) ; V' = V ψ View ; DeleteComp(View)	Delete a view (and its components)
AddToView(View,Comp)	V' ← V ⊗[2] V→(V(View) ∪ {Comp})	Add a component to a view
RemoveFromView(View,Comp)	V' ← V ⊗ V→(V(View) – {Comp})	Remove a component from a view
UpdateAttr(Comp,Attribute,Value)	A' ← A ⊗ Comp→(Attribute→Value)	Update a component attribute
GetAttr(Comp,Attribute,Value)	Value ← A(Comp,Attribute)	Get the value of a component attribute
ApplyOp(Comp,Op,Args)	(C(Comp))(Op))(Args)	Apply a macro operation to a component i.e. use function associated with Op for component kind

Table 1. Fundamental MViews program graph operations.

MViews environments use the display layer of views as editing tools to modify view graphs and thus to modify, indirectly, base graphs. Graphical view editors utilise a direct manipulation interface to modify graphical view components. These modifications are translated into view component operations by generic editing tools supplied by MViews or by application-specific operations associated with the view components. View components are then updated and their view relationships interpret these view component updates and apply appropriate operations to the base components.

MViews supports free-edited textual views made up of one or more textual view components. Each of these view components renders a view component as a sequence of textual characters, called a *text form*, which can be free-edited and parsed to update base component data. A text form corresponds to some significant program entity. For example in SPE, text forms for class interfaces, method implementations, and arbitrary documentation are provided. The corresponding view components are

---

[1]Function domain difference, i.e. the result of F ψ A is the function F with A removed from its domain.

[2]Over-riding union, i.e. the result of F ⊗ A → B is the function F with the mapping for element A of the domain replaced with A → B.

therefore fairly "coarse grain" representations of their program components, with the internal structure of the text forms not being stored directly in the MViews graph structure (although text forms may contain over-lapping information with other components, such as class and feature names and types). This coarse-grained approach is time and space efficient and also allows MViews environments to use conventional text editors rather than structure-editors.

To edit textual views a conventional text editor is used, with an unparser and parser defined. Unparsers convert a base program representation into a textual form and parsers convert an edited piece of text into changes to this program representation. Parsers generate a "parse graph" (effectively a "transient" view graph) which is given to each view component. A view component compares its parse graph data to the base graph and computes and applies required changes to the base graph, reflecting changes made to the textual view. This approach permits free form editing of textual views to be supported. Individual text components can also be structure-edited using menu commands. Text and graphic editors can be tailor-made for an application or specialised from generic MViews tools.

5. Consistency Management in MViews

When a software component is modified all affected views should be updated to reflect the change [26, 30]. This change should also cause language-specific semantics to be rechecked to ensure programmers are informed of errors [5,34]. The change propagation process should be both efficient and as automatic as possible, so programmers need not be concerned with inter-component dependencies [19].

Change propagation is not trivial and a variety of approaches to handling propagated changes is required. Some propagated changes may be applied directly to a view's structure to affect a change. For other changes, it may not be possible to automatically translate them into appropriate view modifications, as there may be an incomplete mapping of concepts between the two views. For example, in SPE, if a change is made to an analysis level view, it may not be at all clear how the modified requirements should be implemented in design or implementation views, and creative input from the programmer is required. In these cases, a visual indication of the change may be appropriate to inform programmers that additional changes need to be made. Lazy application of changes may also be appropriate when a view is hidden [10], and for attribute recalculation for semantic checking where affected values need not be recalculated until required [19].

MViews uses graph relationships not only for structural information but also to describe inter-component dependency for change propagation. Whenever an operation is applied to a graph, the operation generates a description of the change it has made, called an *update record.* This update record is propagated via the graph relationships to components which need to modify their state in response to the original graph update. Components receiving update records are free to interpret them in ways appropriate to the modification. In the rest of this section we formalise the notion of component dependency, update records, and the update record propagation mechanism. In following sections we examine how this mechanism can be used to support a wide variety of view consistency schemes.

5.1. Dependents

An MViews graph is a *dependency graph,* where every component has zero or more related, or *dependent*, components that may be affected by changes to itself. For example, an SPE base class is dependent on its generalisation class, which it inherits features from, and the features it defines, as these determine if the class's interface has been modified. More precisely, the dependents of a component are the other components immediately connected to it via relationships (including the relationship components themselves). For a component, a, the dependents of the component is given by:

$$Dependents(a) \stackrel{def}{=} \{r|r \rightarrow a \rightarrow c \in R\} \cup \{r|r \rightarrow p \rightarrow a \in R\} \cup \{p|r \rightarrow p \rightarrow a \in R\} \cup \{c|r \rightarrow a \rightarrow c \in R\}$$

This mechanism can represent various kinds of software development environment dependencies. For example, a base component has the view components it is linked to as dependents (and vice versa). Components may also be dependent on various aggregate components (and thus have part-of dependency relationships to these components) and may have attribute values dependent on or constrained by other component attribute values (and thus have attribute dependency relationships to these components).

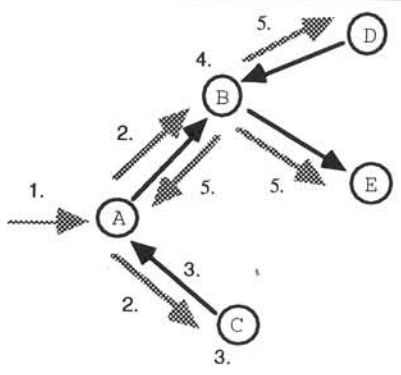
5.2. Update Records and Their Propagation

Update records are generated when graph operations are applied. An update record is, conceptually, a sequence of values of the form `<Component,UpdateKind,Values>`, where:

- *Component*→*Kind*∈*C*
- *UpdateKind* $\stackrel{def}{=}$ *AddComp*|*EstRel*|*CreateView*|*DeleteComp*|*DissolveRel*|*DeleteView*|*AddToView*|*RemoveFromView*|*UpdateAttr*|*String*
- *Values* $\stackrel{def}{=}$ ***seq*** *Value*

For example an *UpdateAttr(**Comp**,Attribute, NewValue)* operation applied to a base component `Comp` generates the update record *<Comp, UpdateAttr, <Attribute, OldValue, NewValue>>*, where *OldValue* is the previous value of *Attribute* for *Comp*.

Each update record generated by an operation on a component is propagated to that component's dependents. Dependents interpret updates and modify themselves (depending on the language structure and semantics for the program under construction). They may, in turn, generate further update records which are propagated. This process is achieved by including appropriate update handlers (similar to event handlers) in components. If an update record is received by a component which matches an available update handler, that handler will be executed. Fig. 4. shows an example of how an operation on a component, A, generates an update record which is propagated to dependents of A: B, C and the relationships A→B and C→A. These dependents determine whether their state should be modified, with B subsequently generating and propagating update records of its own.



1. Operation applied to component A.
2. Update record generated and propagated to A's dependents, B and C and the relationships A→B and C→A.
3. C and the relationships do not change their state.
4. B responds to the update record by applying an operation to update its own state, to be consistent with that of A's.
5. B propagates the update record its operation produced to A, D and E (and the relationships B→A, B→D and B→E) (repeating steps 4, 5 transitively until no more components modify their state).

Fig. 4. An example of update record generation and propagation.

A program's graph, P_1, is thus manipulated by successively applying a sequence of operations $<O_1,...,O_n>$ to $P_1$to form P_{n+1}. Any operation, O_i, applied to a component, C_j, produces a new state, P_i', and propagates an update record, U_o, to the dependents of C_j. Further operations may be generated by dependent component interpretation of these update records to produce the next state, P_{i+1}. The display layer of any views, $\{V_1,...,V_m\}$, containing C_j (or one of its dependents, if updated by U_o) will be re-rendered to reflect the changed program state.

Note, in particular, that the definition of dependents includes inter-view relationships, and hence base components are dependent on their view components (and vice versa). Thus update record propagation proceeds both horizontally, between related components, and vertically, between base and view components.

This dependency graph and update record representation is sufficient to model complex software system component structures and to maintain various semantic constraints between these components.

Components generally define their response to update records in an operational manner using update handlers. As relationships are sent update records in addition to dependent elements, however, relationships can factor out this operational interpretation of update records. MViews component kind specifications can be sub-classed, in a similar manner to object-oriented programming languages, which allows both operations and update responses to be modified for new component kinds.

For example, this mechanism supports the definition of a generic view relationship which always translates updates to attributes on a base component to updates on attributes of view components with the same name (and vice-versa). This behaviour can be modified by defining new view relationships, specialised from the generic view relationship, which support, for example, automatic "expansion" of view components for newly added base components into views. Thus MViews provides the speed and flexibility of an operational approach to the inter-component consistency problem, yet allows high-level relationship-specific constraints to be simply and easily specified.

The description of the MViews consistency model has so far emphasised the underlying graph manipulations involved. In the following two sections we examine the practical application of this model to providing graphical and textual view consistency. A third section examines other applications of update records.

6. Graphical View Consistency

MViews environments use three techniques for keeping graphical view components consistent when their base program components are modified (usually as a result of other view components being updated). The first method is direct update. As graphical view components are rendered directly from their view component structures, they can often be updated directly when the view component receives an update record from its base component.

The second method is expansion and/or disconnection. When a base component is added new graphical view components may be expanded into the view. Similarly, when a base component is deleted, existing view components can be disconnected from the deleted base component and re-rendered to reflect this change.

The third technique supports updates which can not be directly applied to a graphical view. This is where a "fuzzy" relationship exists between information modified in one view and other representations of the modified component in other views.

6.1. Direct Update of Structure

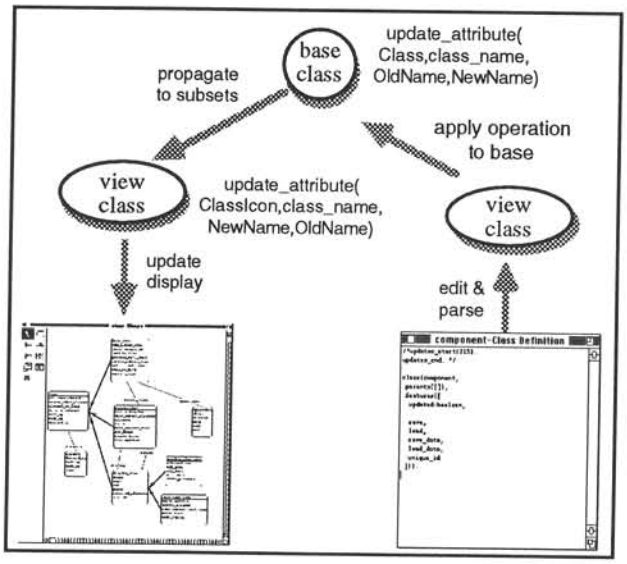


Fig. 5. Direct update of view and graphical view components.

When a view component with a graphical rendering needs to be updated in response to a change to its base component, the view component can usually be updated and re-rendered to reflect the change. For example, if an SPE base class is renamed (i.e. an operation of the form `UpdateAttr(ClassComp, class_name, NewName)` is applied to it) view components of this class can have their `class_name`

attribute modified. The renaming of the class could have originated in a graphical view using direct manipulation or dialogue box, or in a textual view by editing the name and re-parsing the view's text. For example, in Fig. 5. a class is renamed in a textual view resulting in its base class being updated (after parsing). The base class propagates an update record, resulting in its view components being appropriately modified, with view components rendered as class icons redisplaying themselves.

6.2. Expansion and Disconnection

MViews supports three approaches to handling the addition of a new base component where a view is dependent on that addition. The view may:

- Automatically expand the addition (with appropriate layout information and connectivity to existing view components). This is driven by creating new view components in response to `AddComp` or `EstRel` update records, and these new components are linked to existing view and base components as appropriate.
- Indicate to the programmer that there is new information that they may want expanded. When a new base component is added, relationships to existing base components are usually established via a mapping function defined by the view component's kind. Views of these base components can be re-rendered to indicate that new information has been added. The programmer can expand the new component and its relationships at any stage.
- Take no action at all, i.e. the view ignores the update records generated by addition of the new component or relationship.

Similarly, when a base component is deleted, a view may:

- Delete any graphical view components for the deleted base component (with corresponding deletion of connected view components, if appropriate). In this case view components of the base component delete themselves in response to the update record generated. This may also cause appropriate deletion of relationships and dependent components, as necessary.
- Indicate that a view component's base component has been deleted (but not actually delete the view component). Here, a view component simply dissolves its relationship with the deleted base component and re-renders itself to indicate it is no longer connected to a base component. This allows programmers to see view components whose base components have been deleted and update the view appropriately (change or delete glue connections, add new view components to take the deleted component's place, and so on). This contrasts with systems which try to automatically update a view's appearance and composition, often resulting in an undesirable or incorrect layout.
- Take no action.

Different environments may utilise one or more of these approaches to handling additions and deletions in different types of views.

6.3. Storage of Update Records

A third technique for maintaining graphical view consistency is to store update records against base or view components. A built-in component operation, *StoreUpdate(Comp,UpdateRecord)*, is available to implement this. One application of this approach is to support changes propagated over "fuzzy" relationships where automatic modification is not possible, such as the propagation of an analysis level change to a design or implementation level view. In this case, the graphical view visually indicates a change has occured (for example, by re-rendering "updated" components in a different colour). Programmers can then interactively view details of the recorded update record, presented as a list of descriptions of each update in a dialogue, and modify the view appropriately themselves. Using this approach the programmer is informed of changes to base components by the re-rendering of view components and determines and implements appropriate view modifications.

7. Textual View Consistency

Many of the mechanisms described for use in graphical views can also be applied to textual views. As textual view component renderings are text forms, however, the way in which textual views are manipulated and kept consistent differs somewhat from graphical views.

After updating a textual view via free-editing, the view is parsed and appropriate changes made to update base component information. Conversely, when base components are updated, textual view component states can be updated in the same way as graphical view components, as their structure is known and can be manipulated using graph operations. The text forms associated with these view components can not be updated in this manner, however, as their graph structure is not stored directly by MViews in a graph form.

MViews uses an incremental parsing and character substitution technique to update text form contents from an update record. MViews textual views do not store a complete parse tree or token array for text forms but store only the text itself. This requires the environment to incrementally parse a text form and locate the lexical tokens (as their character sequences) which need to be updated, in order to update the view. The advantage of this approach is its space efficiency, and in practice, as the parsing and substitution is performed on small amounts of text (typically < 3K of textual characters), this algorithm is very fast.

This technique has proven very flexible and applicable to a wide range of textual representations. Addition and deletion of information can also be performed in this manner by inserting or deleting character sequences as appropriate. Context-sensitive substitution is possible by locating preceding or following characters. For example, when inserting a new argument to a method, a comma must be added if the new argument is followed by an existing argument. Substitution may be performed multiple times and for different character sequences for the same update record. This is useful when information occurs more than once in a text form (for example, when renaming and/or retyping variables). When a textual view is initially created, MViews environments unparse existing data defined in other views to form the initial contents of the textual view. This initial text may be free-edited or incrementally updated via character substitution, as appropriate.

Many changes propagated to a textual view may not be able to be directly applied to the view using incremental parsing and character substitution. Such updates require the programmer to make the necessary view modifications. When a textual view is selected, any new update records stored against the base components of view components are expanded into a human-readable form. This text is included as part of the view.

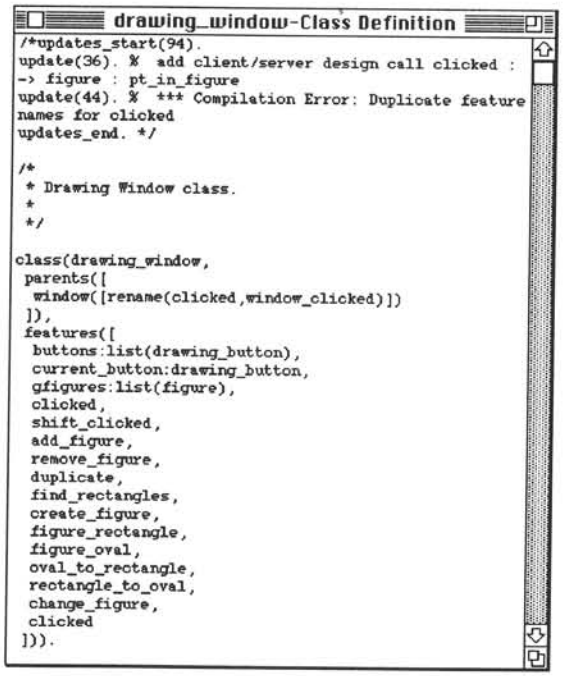


Fig. 6. Updates expanded into a textual view.

Fig. 6 shows a class interface textual view from SPE with two update records expanded in a readable form. The first indicates that addition of a client-supplier relationship, indicating that method *clicked* calls method *pt_in_figure* of class *figure*, has been made in another (graphical) view. SPE cannot automatically change the textual view appropriately because such the connection is implemented in the view as a feature call and the arguments and position of the call cannot be automatically determined. The

update record is expanded, but the programmer is expected to make an appropriate change to the code and remove the update record.

Update records may also be used for processing of errors, as shown by the second update record of Fig. 6. In SPE any semantic errors found when classes are compiled are expanded into textual views using update records (syntax errors are flagged interactively). In the example shown, two features of the same name have been defined in a class. The programmer may also ask SPE to expand all update records into textual views, including those that can be automatically applied to the view's text (such as class and feature renames, feature addition and deletion, changing feature types, etc.). This allows programmers to view all the updates affecting the view before any are applied.

8. Other Applications of Update Records

In addition to view consistency, MViews environments can make use of update records to implement many of the other ISDE facilities described earlier. The following list briefly indicates some of these uses to illustrate the power of the approach:

- **Generic Undo/Redo**: MViews supports a generic undo/redo mechanism by storing update records generated by modified view components as an operation history list. Operation undo is supported by sending stored update records back to their generating components, in reverse order, for reversal. Redo is accomplished by sending stored updates to their generating components, in order, for re-application. A transaction roll-back mechanism is also supported, allowing sequences of operations associated with an aborted editing transaction to be reversed by undoing their effect.
- **Inter-component constraints:** Update records can be used to constrain graphical view component editing. Graphical view components can check update records propagated from related components and reconfigure their position and/or size, or abort the editing operation if the modification is invalid.
- **Incremental attribute recalculation**: Attribute recalculation in other ISDE frameworks is typically driven by state variable dependencies [19, 34]. MViews environments support incremental attribute recalculation driven by update records and attribute dependency relationships. For example, if a new feature is added to a class, the class interface usually must be fully recalculated. SPE can, however, incrementally add the new feature name and type to the class interface attribute list, rather than having to recompute this list from scratch.
- **Update Composition**: Composition of low-level update records into a higher-level update record can reduce the number of update records stored and shown to a programmer. For example, if graphical view components are often dragged to new locations, it is useful to compose update records of the form `UpdateAttr(Comp, depth, OldD, NewD)` and `UpdateAttr(Comp, width, OldW, NewW)` into `ChangeSize(Comp, OldD, OldW, NewD, NewW)`. This new update record then be expanded, stored and automatically applied as one update record. Dependent components may respond to this composed update record or either one of the `UpdateAttr` records (as all three are propagated to dependents).
- **Traceability support**: An analysis component can be related to a more refined form of itself in a design and/or implementation. Updates applied to the analysis component can then be propagated to the design/implementation where they may be stored and unparsed to indicate an analysis change (and possibly a reason for the change). Similarly, this process may be used in reverse to propagate design/implementation changes back to analysis views and components. We are currently extending SPE to support this update record-based consistency management for traceability.
- **Modification Histories and Version Control**: SPE uses recorded update records to create a modification history for components. User defined update records may also be added to document changes that are or have been made. Grouping these stored update records supports version control, as groups of updates can be undone and redone to produce different component versions. We are currently experimenting with version control facilities in MViews environments based on update records and non-sequential undo/redo of these records (to support version merging and partial version reversal).
- **Co-operative Software Development:** Support for multi-user software development allows larger software systems to be constructed but requires support for concurrent view updating [25, 28]. Update records can assist in implementing this. Update records could be broadcast between distributed copies of a software system. Programmers could then work with information being manipulated by other programmers and be informed of the changes to this information by the update records they receive. This would support a degree of collaborative programming with programmers conversing via automatically generated update records. Combined with the version

control facilities, programmers could keep their views of the shared software system consistent with modifications made by other programmers incrementally or in groups.

9. Constructing ISDEs Using the MViews Model

9.1. Design

A textual notation, called MVSL, has been developed for expressing base and view component structure, operations and update record responses during preliminary design of MViews-based environments. This model has been formally defined using operational semantics and implemented in Haskell to ensure MViews structures and operations have a precise meaning. A complementary visual notation, called MVisual, is used to define the appearance and behaviour of a view's display layer and user interface. MVSL and MVisual specifications for an environment conceptually communicate via event flows represented as update records. For more detail on these notations, consult [17].

More detailed design is done using an object-oriented architecture based on the MViews abstractions. Fig. 7. shows the hierarchy of classes defined by this architecture. Components are modelled as classes (for example, a base layer is `base_layer` and a graphic icon is `graphic_icon`). Component attributes are represented by objects associated with a `component` and relationships are defined as attributes which refer to relationship component objects.

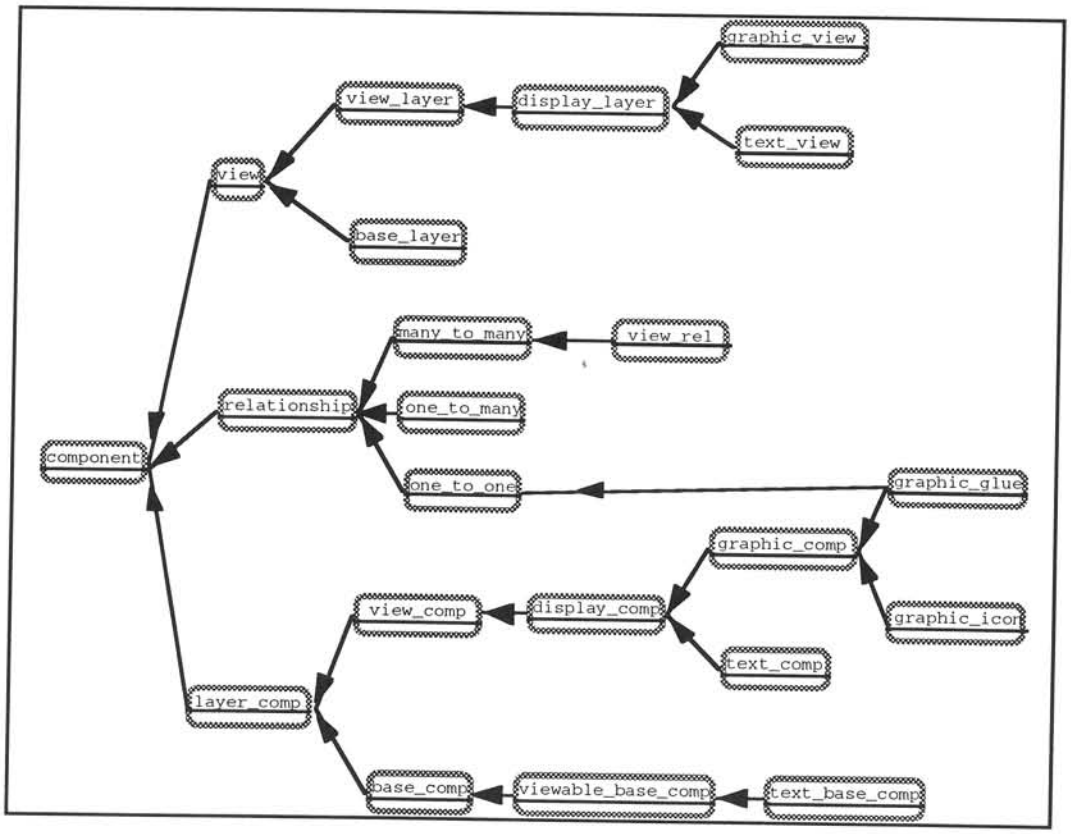


Fig. 7. An object-oriented hierarchy of MViews components (generated from an SPE view).

Environment-specific components are defined by specialising classes appropriately. For example, a `class_icon` for SPE can be defined by specialising `graphic_icon` from the MViews framework and defining appropriate extra attributes (for example, `class_name` and `feature_names`) and methods (for example, `update_attribute` for `class_name` and `add_feature_name` for `feature_names`).

Each kind of MViews component class defines the operations it supports as methods and the component-specific data as attributes. The `component` class abstracts out all common component behaviour such as attribute updating and storage, update record generation and propagation and participation in relationships. Layer components extend `component` data and operations for different kinds of layer components (base, view and display). Relationship components link other components and may define attributes and operations of their own. Views group graph components and provide additional operations for manipulating these graphs. The view/display layer relationship is modelled by

inheritance (i.e. a `display_comp` inherits all the information of a `view_comp` and hence one view component object incorporates both view and display layer data and behaviour).

9.2. Implementation

MViews has been implemented as a reusable framework of Snart[3] classes. Each kind of component is implemented using a Snart class with component attributes described as Snart object attributes and operations implemented as methods. Update records are represented by Prolog terms. Methods used to process these update record terms are written in a declarative style by specifying update record patterns (predicate head) to respond to and the actions to take for each kind of update record (body).

Graphical view components are updated by modifying their view component state in response to an update record. The component is redrawn to reflect the change, with MViews supporting abstractions for updating graphical view components incrementally for efficiency. The framework also provides building-blocks for constructing graphical editors, and for describing graphical view component behaviour and common graphical view metaphors:

- icon/connector and icon/sub-icon approaches;
- manipulation mechanisms, such as click and drag and tool palettes;
- hypertext-like click-points on icons; commonly used graphical component constraints;
- and lazy update record processing and update record composition.

The incremental parsing and token substitution algorithm for updating textual views is implemented in Prolog. Regular expressions are used to specify the character sequences (i.e. lexical tokens) to locate. These are returned as their character value with their offsets (start position, length) in the view text. Textual view components define how update records are applied to their textual representation using methods written in a declarative style, similar to update record response methods. MViews uses a standard text window editor for displaying and editing textual views. This editor is augmented with menus to assist view navigation and update record management within an environment.

Software systems and their views need to be stored between invocations of an environment. The MViews framework provides software system persistency support that is efficient in both time and space, requires little or no application-specific programming to support, and is flexible enough to suit the requirements of different environments. MViews supports component persistency via class methods or transparently via persistent Snart objects. We have found the second approach to be much more natural, easy-to-use and manageable and are currently improving the performance of Snart persistency to support distributed object stores. This will permit us to implement a multi-user form of MViews supporting collaborative software development.

An object-oriented Prolog proved to be a good choice of prototyping language for MViews. Update record processing using a declarative style of `update record pattern->response` forms a natural approach to specifying component responses to update records, how update records are unparsed, and how to apply update records to update textual views and view components. Representing update records as Prolog terms provides an efficient implementation for generating, propagating and storing update records.

10. Experience with MViews

MViews has been used to model, design and implement a number of diverse environments and applications. The first application of MViews was in the development of the Snart Programming Environment (SPE), used as an example throughout the earlier discussion. Other environments developed include an Entity-Relationship modeller, a dialogue box designer, and a program visualisation system. Here we briefly describe these systems to illustrate the reusability of the MViews framework, with an emphasis on our reuse of MViews dependency graphs and the multiple textual and graphical views provided by each system.

---

[3]Snart is the Object Oriented Prolog that SPE provides a programming environment for.

10.1. MViewsER

Entity-relationship (ER) modelling [7] is typically used to model database systems by decomposing data into entities and relationships between entities. ER models are translated into relational database schemas (RDSs) to provide a data model for information systems. One solution to integrating ER and RDS specification is to provide graphical ER modelling views and complementary textual RDS views, with consistency management between the two. Fig. 8. shows an example of MViewsER, which takes this approach to database model specification.

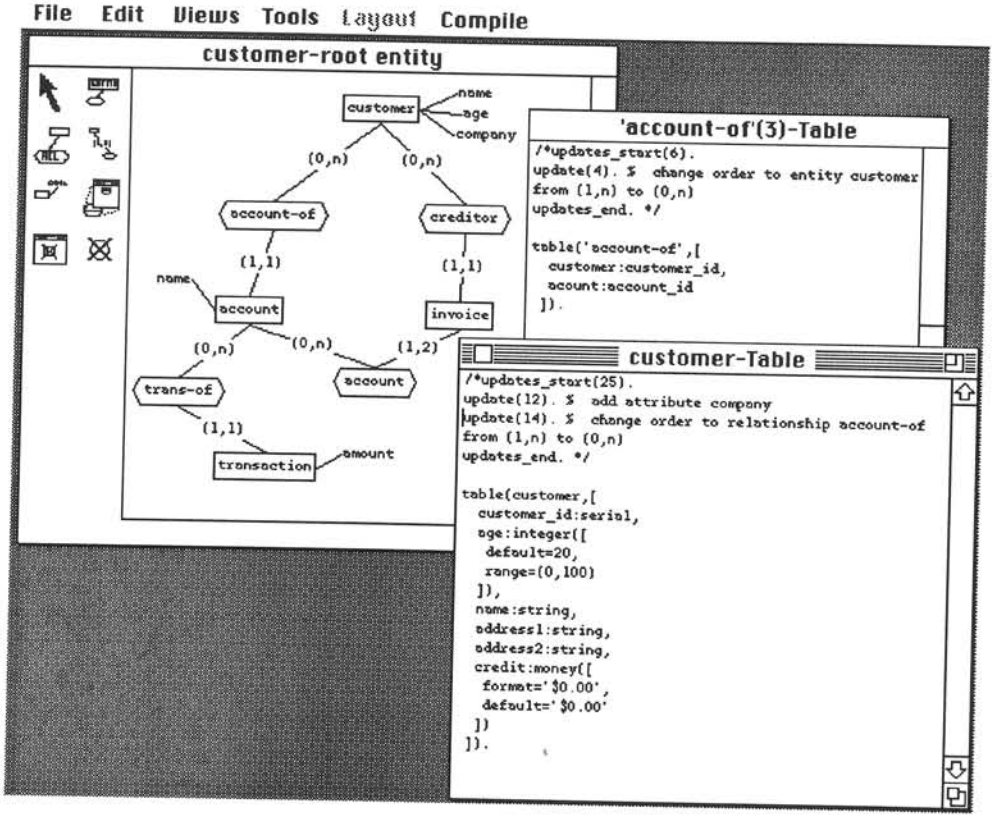


Fig. 8. MViewsER graphical ER views and textual RDS views.

MViewsER supports graphical ER diagram views and textual RDS views. The latter contain a table definition including table field names, field types, and field attributes. RDS views are parsed to update table information. The graphical ER views provide a high-level specification system with details about RDS requirements ignored. Textual RDS views can be generated from ER data and extra information added about field types, defaults, range values and so on.

Consistency management is employed between ER diagrams and RDS tables. ER diagram views are updated directly by changes to RDS table views. RDS views are updated by unparsing update records (as shown in Fig. 8.) in a similar manner to SPE. Some update records can be automatically applied to reflect changes to entities and relationships. Other update records serve as documentation to inform programmers of ER model changes that may or may not impact on the RDS tables. ER models and RDS tables are represented as base layer graphs, and views of these structures and rendered and manipulated using appropriate view abstractions.

10.2. MViewsDP

MViewsDP is a dialogue painter for specifying Macintosh-style dialogue boxes. MViewsDP provides a graphical view which allows dialogue box components to be interactively added, deleted and modified. This view shows the form a dialogue box will have when actually used. One or more textual views are used to specify additional information about the dialogue box. These contain a Prolog predicate defining the dialogue box's sub-components and predicates used to set up initial values for fields, check the validity of entered data, and carry out any processing of entered data for passing back to Prolog predicates which invoke the dialogue box. Graphical and textual views are kept consistent with update records and an update history is kept. Graphical view components are redisplayed after receiving

updates while textual views unparse update records. Fig. 9. shows an example of MViewsDP views and a resulting dialogue box.

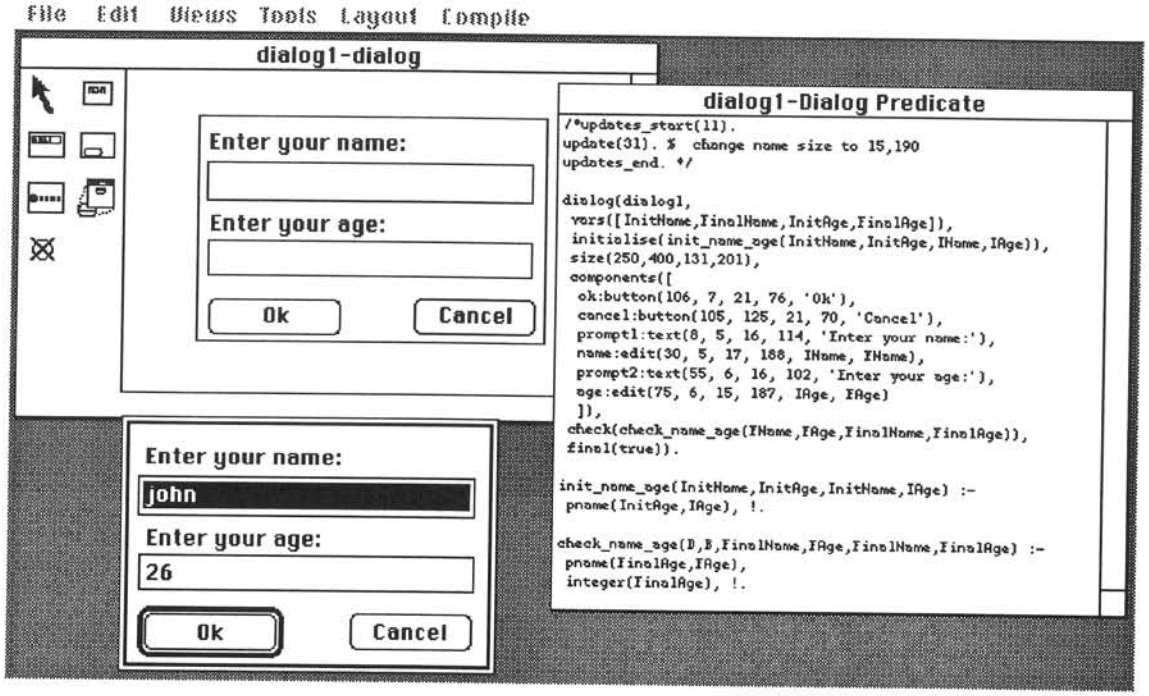


Fig. 9. An example of MViewsDP views and a corresponding dialogue box.

A difference between MViewsDP and both SPE and MViewsER is that graphical dialogue box sub-components must be enclosed by their owning dialogue box's border and are displayed relative to their owner's location. Sub-component icons are also shifted and resized when their owner's border is shifted or resized. These sub-component icons are thus dependent on updates on their enclosing dialogue, i.e. they implement a container-containment approach, rather than an icon-connector model. The dialogue components and their views are represented as MViews graphs. Update record composition reduces the number of update records used and graphical component constraints are employed to enforce dialogue component containment, scaling and re-location when they or their enclosing dialogue border is manipulated.

10.3. CernoII

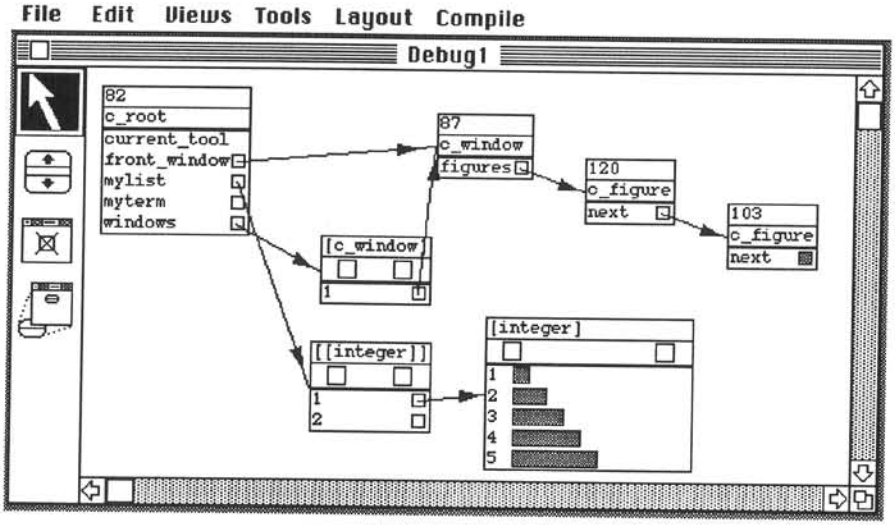


Fig. 10. An example of CernoII multi-object views.

Visual debugging allows executing programs to be debugged by displaying object data, relationships and control flow [12]. CernoII, a visual debugging system for SPE, reuses MViews to build graphical debugging views for Snart programs [13]. Snart provides a dynamic object tracing mechanism where individual objects and an object's features can be spied. CernoII uses this to produce update records

equating to object method calls and attribute assignment which are used to drive program visualisations. CernoII defines views and view components to describe the state of an executing Snart program. A multi-object view shows one or more Snart objects and references to other Snart objects. Programmers can specify which attributes of an object are shown for each object view component. References are expanded by programmers and object attribute values can be interactively modified by the user. Fig. 10. shows an example of a Cerno multi-object view.

10.4. Other Systems

Many other uses of MViews have been developed or are under development. Specialisations of SPE and Cerno are being used for constructing building models [1], and for implementing an environment for developing EXPRESS programs and corresponding EXPRESS-G diagrams[4] [2]. A generic dataflow diagrammer is under development and is being specialised for the following applications: a high-level DFD diagrammer for analysis and more detailed diagrams for dataflow between features in SPE; a visual dataflow programming language, similar to Prograph [9], which will give a complementary way of defining methods to textual Snart code in SPE; and a DFD-based concurrent programming language. An environment for programming Hyper-Pascal, a visual Pascal-like language is also being implemented using MViews [24]. MViews also provides a mechanism for implementing tool based abstraction, as suggested by [14]. Components represent tools, relationships are used to connect tools and update records describe the exact change to a tool's data structures.

11. Discussion

Development of SPE, MViewsER, MViewsDP and CernoII has shown that MViews greatly assists the development of integrated software development environments. For example, MViewsER and MViewsDP took less than a person week each to develop from initial specification to final implementation. Use of MViews for the development of CernoII has resulted in a much faster development time, less errors during implementation and much improved functionality and extensibility over an earlier prototype which did not use MViews [13]. Similarly, development of SPE has been greatly enhanced, in terms of development time, extensibility and useful functionality, compared with an earlier object-oriented programming environment [16].

In the rest of this section we discuss how the key abstractions of MViews make the task of building integrated environments easier in comparison with existing ISDE frameworks.

11.1. Software System Data Representation

MViews' graph-based model for representing software system structure and semantics has proved very flexible for a diverse range of applications. Graph-based visual languages, such as class relationship diagrams and entity-relationship models, can be naturally defined as graph components and relationships. Hierarchical language structures, such as class interfaces and dialogue box components, can be represented equally well using part-of relationships between components. Representation of language semantic information using the same technique allows this information to be rendered (and sometimes manipulated) in the same manner as structural components.

MViews' representation model compares favourably with other approaches. It is more flexible than tree-based environments, such as the Synthesiser Generator [34], and a more homogeneous model than extended tree-based representations for graphical language support, such as LOGGIE [5]. MViews supports a more language-oriented representation model than that of Dora [29], with similar capabilities to the graph-based representation of GLIDE [22], although MViews graph components can be specialised to support software system evolution (see later). While not as expressive as SPGs [27], MViews' graph representation is much simpler and is reuseable for view component specification.

11.2. Multiple Textual and Graphical Views

MViews supports multiple textual and graphical views of software system data via views and textual and graphical renderings of these views. As view layer components are graph components they share a

---

[4] EXPRESS and EXPRESS-G are object-oriented specification languages [21].

common representation and manipulation technique with base layer graphs. This aids in the definition of view layer components based on base layer structure and semantic information.

Display layers provide a rendering of views and an interactive editor for modifying view components. The distinction between display and view layers is useful for modelling purposes as it allows view data to be defined using the graph model and a view rendering/interaction mechanism to be defined for the information in this view graph. The provision of a set of building blocks for view editors in the MViews framework makes new environment views easier to build, in a similar way that Unidraw makes drawing editors easier to build [36].

11.3. Consistency Management

Update records provide several advantages over MVC-style update messages, as used in Smalltalk [15], and state variable change propagation, as used by Unidraw [36] and active databases [19]. Base and view component updates can be propagated using update records without the need for intervention from a controller or editor component. This allows a more modular approach to the propagation of component change.

As MViews uses relationships to determine dependency, it does not require a separate dependency graph network in addition to component data, as do other approaches, such as [40]. MViews uses structural relationship components to not only link other components but also to form the dependency graph for a program. This also means update records do not need to store additions and deletions of dependents like change reports [40], as these are encoded by separate `EstRel` and `DissolveRel` operation update records. Unlike the `ItemList` structure [10], MViews can represent a much richer set of structures using graphs. As updates are recorded sequentially, redundant copies of `Item` (attribute) values are not stored, resulting in a simpler undo/redo mechanism. As update records can be stored and manipulated independently of their generating components, they support environment facilities in a more homogeneous way than database transactions and object dependency links, as used in Garden [31].

As an update record documents the exact change a component has undergone, dependents of the component can update themselves in an incremental manner. For example, a view component can determine the exact change to one of its base component's attributes or a change in a related view component. This can be used to implement efficient incremental update and redrawing of a view component from changes to its base component's state. This mechanism can also provide a flexible constraint mechanism between components. Lazy update record processing proved very useful for composing update records into more abstract records and for lazy view consistency and constraint maintenance in MViewsER and MViewsDP. This lazy update record processing is quite low-level, however, and only supported by the Snart framework. More powerful methods of specifying update record composition and lazy processing are required, particularly support for these at the modelling and architecture levels. Dependent component attribute recalculation could similarly do with more abstract specification and better framework support.

Textual views are kept consistent by unparsing and automatically applying a sequence of update records. Parsing a textual view updates appropriate base program information and thus affects other views. The incremental parsing and token substitution model allows many textual view components to be automatically updated by MViews environments. This model is more flexible than the FormsVBT approach [4] as it allows concurrent textual and graphical view updates and supports unrestricted free-editing of text. Non-uniquely identifiable structures can't be automatically updated using this model (such as documentation or method implementation code) but the MViews framework does support annotation of view text which allows it to be uniquely identified. An improved text editor could be used to do this annotation more transparently, similar to the FIELD editor [33].

Update records could be used in structure-oriented environments, such as Mjølner [25] or Dora [29], to achieve view consistency and other environment tasks. Structure-oriented editing commands could generate update records in the same manner as MViews graph operations (which are, in fact, structure-oriented). These update records could be propagated, stored and unparsed to achieve the same kind of view consistency mechanisms as provided by MViews environments. It may also be possible to make more automatic updates as the structure of all views is always known and manipulable.

11.4. Tool Integration and Extensibility

MViews uses a canonical base program representation with multiple views of this representation for different environment tools. A chief advantage of MViews is that its components can be specialised and extended via inheritance to support software system evolution, thus partially solving a major problem with integrating multi-view editing environments [26]. For example, SPE is actually specialised from IspelM, a generic multi-view programming environment for any (class-based) object-oriented language. IspelM is itself built from a collection of component classes specialised from MViews. SPE extends IspelM for Snart programming by further specialising the IspelM framework. If SPE structures need to be modified (for example, to extend the environment tools), then these structures extended by specialising the current SPE can still use existing environment data.

MViews supports tool data integration via the canonical base program view with different kinds of views of this representation for different tools. User interface integration is supported via views built from a common set of window, dialogue and menu building blocks. Views can also be used to integrate tools not built using MViews by parsing and unparsing external tool data formats and linking these to base components via view components (where applicable). Update records can also be generated in response to external tool operations and be used to generate corresponding external tool operations (where the external tool supports an appropriate interface).

11.5. Persistency and Multi-user Software Development

Initial versions of the MViews framework provided a collection of methods for saving and loading component state, similar to the mechanism used by Unidraw [36]. This mechanism proved to be less than ideal. It is clumsy and error prone as it requires significant programming effort and is difficult to update when an environment's component structures change. Dora uses PCTE to store program data [29], GARDEN uses an object-oriented database [31], and [28] uses a distributed Smalltalk, but these all require some form of database views, which can be very problematic when maintaining an integrated environment [26, 27]. Snart supports transparent object persistency and we have used this in later versions of MViews to support component persistency. This has proved to be a much more satisfactory approach which preserves our canonical data representation scheme while supporting flexible software system data persistency.

MViews does not currently support multi-user software development. We are extending it to support flexible version control based on update records and are extending Snart to support distributed persistent object stores. We also plan to broadcast update records between environments to support collaborative software development, particularly for analysis and design views.

12. Conclusions and Future Research

MViews provides a novel model for ISDEs that support multiple textual and graphical views of information with consistency management. MViews provides a model based on dependency graphs for representing program data and views of this program data. Views are rendered graphically or textually with graphical views interactively edited and textual views free-edited and parsed. The novel update record mechanism is used for a variety of environment consistency requirements. Examples include maintaining textual and graphical view consistency in a novel manner, propagating changes between related components for attribute recalculation and enforcing constraints, supporting undo and redo of editing operations, and supporting component change documentation.

MVSL and MVisual support the specification of environments based on the MViews model. The MViews object-oriented architecture and Snart framework allow these environments to be implemented much more easily than without MViews' abstractions and building blocks. Dependency graph components are implemented as specialisations of framework classes. Methods for processing update records, generated, represented and propagated efficiently as Prolog terms, are implemented in a declarative style. MViews has been reused to produce SPE, a novel ISDE for constructing Snart software; an entity-relationship modeller with textual relational database views; a dialogue painter with textual constraint specification views; and various program visualisation views for SPE.

We are currently extending MViews to support multi-user, distributed software development. This includes use of update records for version control, configuration management, and collaborative software development where update records are broadcast between different environment invocations. The MViews framework is also being extended to provide better support for attribute grammars to

support efficient recalculation of static language semantic values, more abstract specification of update record composition and lazy update record propagation, and support for non-sequential undo/redo for version control. SPE is being extended to support collaborative, distributed object-oriented software development using these MViews facilities. We are planning to experiment with partial generation of MViews environments from MVSL and MVisual and to provide visual specification of view appearance and functionality.

Acknowledgments

John Grundy has been supported by an IBM Postgraduate Scholarship, a William Georgetti Scholarship and a New Zealand Universities Postgraduate Scholarship. We also gratefully acknowledge the financial support of the University of Auckland Research Committee and the many helpful comments provided by our colleague Rick Mugridge in the preparation of this paper.

References

[1] Amor, R., and Hosking, J.G. Multi-disciplinary views for integrated and concurrent design, in Mathur, K.S., Betts, M.P., and Tham, K.W. Ed.s, Selected (refereed) papers from the Proc of the First International Conference on Management of Information Tecnology for Construction, Singapore, August 1993, 255-267.

[2] Amor, R., Augenbroe, G., Hosking, J., Rombouts, W., and Grundy, J. Directions in modelling environments, TU Delft Dept of Civil Engineering working paper, 1993.

[3] Arefi, F., Hughes, C.E., and Workman, D.A. Automatically Generating Visual Syntax-Directed Editors, Communications of the ACM 33, 3 (March 1990), 349-360.

[4] Avrahami, G., Brooks, K.P., Brown, M.H. A Two-View Approach to Constructing User Interfaces, ACM Computer Graphics 23, 3 (1990), 137-146.

[5] Backlund, B., Hagsand, O., Pehrson, B. Generation of Visual Language-oriented Design Environments, Journal of Visual Languages and Computing 1, 4 (1990), 333-354.

[6] Brown, M.H. Zeus: A System for Algorithm Animation and Multi-View Editing, 1991 IEEE Symposium on Visual Languages, IEEE Computer Society Press, 4-9.

[7] Chen, P.P. The Entity-Relationship Model - Toward a Unified View of Data, ACM Transactions on Database Systems 1, 1 (1976) 9-36.

[8] Coad, P., Yourdon, E. Object-Oriented Analysis, Second Edition, Yourdon Press, 1991.

[9] Cox, P.T., Giles, F.R., Pietrzykowski, T. Prograph: a step towards liberating programming from textual conditioning, 1989 IEEE Workshop on Visual Languages, IEEE Computer Society Press, 150-156.

[10] Dannenberg, R.B. A Structure for Efficient Update, Incremental Redisplay and Undo in Graphical Editors, Software-Practice and Experience 20, 2 (February 1990), 109-132.

[11] Dart, S.A., Ellison, R.J., Feiler, P.H., Habermann, A.N. Software Development Environments, COMPUTER 20, 11 (November 1987), 18-27.

[12] Fenwick, S. and Hosking, J.G. Visual Debugging of Object-Oriented Systems, Departmental Report #65, Department of Computer Science, University of Auckland, Auckland, New Zealand, 1993.

[13] Fenwick, S. A Visual Debugger for Object-Oriented Programs, MSc Thesis, Department of Computer Science, University of Auckland, Auckland, New Zealand, 1994.

[14] Garlan, D., Kaiser, G.E, Notkin, D. Using Tool Abstraction to Compose Systems, COMPUTER 25, No. 6 (June 1992), pp. 30-38.

[15] Goldberg, A., Robson, D. Smalltalk-80: The Language and its Implementation, Addison-Wesley, Reading, MA, 1984.

[16] Grundy, J.C., Hosking, J.G., Hamer, J. A Visual Programming Environment for Object-oriented Languages, Proceedings of TOOLS US '91, Prentice-Hall, August 1991, 129-138.

[17] Grundy, J.C. Multiple textual and graphical views for interactive software development environments, PhD Thesis, Department of Computer Science, University of Auckland, 1993.

[18] Grundy, J.C. and Hosking J.G. Integrated Object-oriented Software Development in SPE, Procs 13th New Zealand Computer Society Conference, Auckland, August 1993.

[19] Hudson, S.E. Incremental Attribute Evaluation: A Flexible Algorithm for Lazy Update, ACM TOPLAS 13, No. 3 (July 1991), 315-341.

[20] Ingalls, D., Wallace, S., Chow, Y.Y., Ludolph, F., Doyle, K. Fabrik: A Visual Programming Environment, Proceedings of OOPSLA '88, ACM Press, 176-189.

[21] ISO/TC184 Part 11: The EXPRESS Language Reference Manual in Industrial automation systems and integration - Product data representation and exchange, Draft International Standard, ISO-IEC, Geneva, Switzerland, ISO DIS 10303-11, August 1992.

[22] Kleyn, M.F., Browne, J.C. A High Level Language for Specifying Graph Based Languages and their Programming Environments, Proceedings of the 1993 IEEE International Conference on Software Engineering, May 1993, 324-334.

[23] LPA. LPA Prolog Version 4.5 Reference Manual, Logic Programming Associates, London, 1993.

[24] Lyons, P., Simmons, C., Apperley, M. HyperPascal: Using visual programming to model the idea space, Procs 13th New Zealand Computer Society Conference, Auckland, August 1993, 492-508.

[25] Magnusson, B., Bengtsson, M., Dahlin, L., Fries, G., Gustavsson, A., Hedin, G., Minör, S., Oscarsson, D., Taube, M. An Overview of the Mjølner/ORM Environment: Incremental Language and Software Development, Proceedings of TOOLS '90, Prentice-Hall, pp. 635-646.

[26] Meyers, S. Difficulties in Integrating Multiview Editing Environments, IEEE Software 8, 1 (January 1991), 49-57.

[27] Meyers, S. Representing Software Systems in Multiple-View Development Environments, PhD Thesis, Department of Computer Science, Brown University, CS-93-18, 1993.

[28] Nascimento, C., and Dollimore, J. A model for co-operative object-oriented programming, IEE Software Engineering Journal 8, 1 (1993), 41-48.

[29] Ratcliffe, M., Wang, C., Gautier, R.J., Whittle B.R. Dora - a structure oriented environment generator, IEE Software Engineering Journal 7, 3 (1992), 184-190.

[30] Reiss, S.P. PECAN: Program Development Systems that Support Multiple Views, IEEE Transactions on Software Engineering 11, 3 (1985), 276-285.

[31] Reiss, S.P. Working in the GARDEN Environment for Conceptual Programming, IEEE Software 4, 11 (November 1987), 16-26.

[32] Reiss, S.P. Connecting Tools Using Message Passing in the Field Environment, IEEE Software 7, 7 (July 1990), 57-66.

[33] Reiss, S.P. Interacting with the Field environment, Software practice and Experience 20, S1 (June 1990), S1/89-S1/115.

[34] Reps, T., Teitelbaum, T. Language Processing in Program Editors, COMPUTER 20, 11 (November 1987), 29-40.

[35] Symantec Corporation. THINK C Reference Manual, 1990.

[36] Vlissides, J.M. Generalized Graphical Object Editing, PhD Thesis, Stanford University, CSL-TR-90-427, 1990.

[37] Wasserman, A.I., Pircher, P.A. A Graphical, Extensible, Integrated Environment for Software Development, SIGPLAN Notices 22, 1 (January 1987), 131-142.

[38] Welsh, J., Broom, B., Kiong, D. A Design Rationale for a Language-based Editor, Software - Practice and Experience 21, 9 (1991), 923-948.

[39] Whittle, B.R., Gautier, R.J., Ratcliffe, M. Trends in Structure Oriented Environments, Technical Report UCW-SEG-601-92, University Colledge of Wales, Abersystwyth, 1992.

[40] Wilk, M.R. Change Propagation in Object Dependency Graphs, Proceedings of TOOLS US '91, Prentice-Hall, August 1991, 233-247.